Reconfiguración Dinámica de Arquitecturas Software Aplicada a la Tolerancia a Fallos

Cristóbal Costa-Soria[1], Jennifer Pérez[2], Jose A. Carsí[1],
Diego Alonso[3], Francisco Ortiz[3], Juan Ángel Pastor[3]

[1] Dpto. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Cno. de Vera s/n, 46022 Valencia
ccosta@dsic.upv.es, pcarsi@dsic.upv.es
[2] E.U. Informática, Universidad Politécnica de Madrid, Ctra. Valencia km 7, 28051 Madrid
jenifer.perez@eui.upm.es
[3] Depto. Tecnología Electrónica, Universidad Politécnica de Cartagena, Campus Muralla del Mar s/n, 30202 Cartagena
diego.alonso@upct.es, francisco.ortiz@upct.es, juanangel.pastor@upct.es

Resumen

En la actualidad, el desarrollo de sistemas software tolerantes a fallos se realiza a un nivel dependiente de la tecnología, con lo que aumenta su complejidad y disminuye la reutilización. La mayoría de estrategias de tolerancia a fallos son estáticas: se basan en replicar elementos críticos para que, ante cualquier fallo, sus réplicas tomen el relevo. En este trabajo se describe cómo la reconfiguración dinámica de arquitecturas software puede aplicarse para desarrollar sistemas tolerantes a fallos. Las técnicas de reconfiguración dinámica permiten cambiar la configuración de sistemas software complejos en tiempo de ejecución, sin necesidad de detener el sistema. Este artículo describe cómo la reconfiguración dinámica es soportada a nivel de arquitecturas software y mediante aspectos, separando la funcionalidad de reconfiguración –y las políticas de recuperación frente a fallos- del resto de funcionalidades del sistema. Esto se ilustra mediante la definición de las políticas de recuperación del sistema de visión del Agrobot, un sistema robótico del ámbito agrícola.

1. Introducción

Hoy en día, la mayoría de sistemas software son complejos, distribuidos y dinámicos. Esto ha hecho que la flexibilidad, seguridad y tolerancia a fallos sea un factor fundamental en su proceso de desarrollo. La tolerancia a fallos es un aspecto ampliamente cubierto en el área de los sistemas robóticos, pero esto se realiza a nivel de implementación, con lo que las soluciones son dependientes de tecnología y poco reutilizables.

Existen numerosas estrategias de diseño y patrones arquitectónicos para la detección y tolerancia a fallos [20]. Estas estrategias se basan en el uso de diferentes tipos de redundancia, que se caracterizan por tener varias réplicas de los elementos críticos ejecutándose concurrentemente en el sistema. En el caso de fallar uno de estos elementos, su funcionalidad continúa siendo ofrecida gracias al resto de réplicas en ejecución, que permiten "ocultar" el fallo. Este enfoque clásico de redundancia presenta la ventaja de que el tiempo de recuperación frente al fallo es inmediato, puesto que las réplicas ya se encuentran en ejecución y tienen el mismo estado que el elemento que ha fallado, y que la sustitución del elemento defectuoso es sencilla.

Sin embargo, este enfoque presenta una serie de inconvenientes: (i) los puntos de sincronización donde se comparan los resultados de las réplicas son difíciles de definir y de implementar; (ii) las réplicas consumen recursos del sistema; (iii) la complejidad del sistema aumenta, debido a que el número de mensajes entre las diferentes partes del sistema crece rápidamente con el número de réplicas; y (iv) este enfoque es estático, no puede adaptarse ante cambios del sistema durante su ejecución.

Una alternativa al uso de la redundancia para soportar tolerancia a fallos es el uso de técnicas de reconfiguración dinámica [32]. La reconfiguración dinámica permite dotar de flexibilidad a un

sistema software, permitiendo tanto cambiar su configuración como sustituir unos elementos por otros en tiempo de ejecución. El uso de la reconfiguración dinámica en el diseño de sistemas tolerantes a fallos permitiría evitar tanto la complejidad de mantener operativos de forma simultánea a varios componentes redundantes, como el consumo de recursos de computación adicionales.

Sin embargo, el desarrollo y mantenimiento de sistemas reconfigurables es complejo y costoso. Por un lado, su desarrollo es complejo ya que los mecanismos de reconfiguración son dependientes de la tecnología [24]: dependiendo de la tecnología empleada y de cómo se desarrolle el sistema, la forma de detener de un modo seguro partes del sistema variará, así como el modo de cambiar su configuración. Por otro lado, el mantenimiento es costoso, ya que las especificaciones de reconfiguración (i.e. las políticas de reconfiguración) suelen estar entremezcladas con la funcionalidad del sistema. Esto es debido a que la ejecución de la funcionalidad del sistema es la que genera la necesidad de reconfiguración, y por tanto, a través de la funcionalidad se detecta cuándo debe reconfigurarse el sistema.

Es por esto que, para facilitar el desarrollo y mantenimiento de sistemas dinámicamente reconfigurables: (i) la reconfiguración debe describirse a un mayor nivel de abstracción, independientemente de los detalles específicos de tecnología (e.g. los mecanismos que soportan la reconfiguración del sistema en tiempo de ejecución) y, (ii) la funcionalidad del sistema debe estar claramente separada de la especificación de cómo debe reconfigurarse el sistema en tiempo de ejecución. Para ello, pueden seguirse dos enfoques complementarios: las arquitecturas software para describir sistemas software a un alto nivel de abstracción, y el desarrollo orientado a aspectos para separar la reconfiguración de la funcionalidad del sistema.

Las arquitecturas software [29] describen un sistema en términos de elementos arquitectónicos (componentes y conectores) y las interacciones entre sí (conexiones). La reconfiguración dinámica de arquitecturas permite cambiar la composición de un sistema en ejecución en términos de creación/destrucción de instancias de elementos arquitectónicos y de la modificación de las conexiones entre ellos [15],[23]. Por otro lado, el Desarrollo de Software Orientado a Aspectos (AOSD) [22] propone la separación de los *crosscutting-concerns*[1] de un sistema software en entidades separadas denominadas aspectos. La separación de dichos *crosscutting-concerns* en entidades separadas permite su reutilización en distintas entidades del sistema software a la vez que facilita su mantenimiento. La reconfiguración dinámica es intrínsecamente un *crosscutting-concern*: las reglas de reconfiguración están mezcladas con el código funcional y se encuentran diseminadas por todo el sistema. Por esta razón, la encapsulación del código de reconfiguración en aspectos, ayudará al desarrollo de sistemas reconfigurables a la vez que facilitará su mantenimiento.

En este trabajo se describe cómo pueden definirse estrategias de recuperación frente a fallos: (i) a través de la reconfiguración dinámica, evitando el uso excesivo de elementos redundantes y optimizando el uso de recursos; (ii) a un alto nivel de abstracción, mediante arquitecturas software, contribuyendo a reducir la complejidad del sistema; y (iii) mediante aspectos, separando la funcionalidad de reconfiguración/ recuperación del resto de la funcionalidad del sistema software, con lo que se aumenta la reutilización de dichos artefactos. Para ello, se ha elegido el modelo PRISMA, el cual permite especificar arquitecturas software orientadas a aspectos independientes de plataforma y da soporte al Desarrollo Dirigido por Modelos, generando código C# a partir de sus arquitecturas [26], [27].

Las ideas presentadas en este trabajo se ilustran con un caso de estudio del dominio robótico. El motivo de elegir este dominio es debido a que, aunque la robótica engloba múltiples disciplinas (como la mecánica, la electrónica y la informática), la capacidad de un robot para actuar de forma autónoma e *inteligente* reside cada vez más en su software de control. Por ello, para superar las limitaciones de los robots es imprescindible superar las limitaciones del software que gobierna su comportamiento. Para manejar esta complejidad es de gran ayuda el uso de las herramientas y metodologías proporcionadas por la Ingeniería del Software.

[1] Funcionalidad transversal de un sistema: auditoría, persistencia, seguridad, comunicación distribuida, etc.

La estructura del artículo es la siguiente: en la sección 2 se introduce PRISMA y los aspectos que ofrecen soporte a la reconfiguración dinámica de arquitecturas. En la sección 3 se introduce el Agrobot y el diseño de una estrategia de recuperación mediante nuestra propuesta. Finalmente, en las secciones 4 y 5 se describen los trabajos relacionados y las conclusiones.

2. Reconfiguración dinámica de Arquitecturas Software

El proceso de reconfiguración dinámica puede ser de dos tipos, según como se inicie dicho proceso [15]: *ad-hoc*, si el proceso de reconfiguración es iniciado y dirigido en tiempo de ejecución por un agente externo al propio sistema; o *programado*, si el proceso de reconfiguración es iniciado y dirigido en tiempo de ejecución por el propio sistema, en base a un conjunto de reglas de reconfiguración definidas en el sistema. La reconfiguración dinámica programada puede usarse para describir sistemas auto-organizados, i.e, sistemas capaces de reconfigurar su topología por sí mismos en tiempo de ejecución. En particular, la reconfiguración programada puede usarse para describir sistemas tolerantes a fallos, a través de la definición de las estrategias de recuperación a realizar tras detectar un fallo.

En un sistema software complejo, como es el caso de los sistemas robóticos, la reconfiguración dinámica programada puede ser necesaria, además de para el propio sistema, para cada uno de los subsistemas que lo forman (i.e. componentes complejos). Para ello, puede seguirse un enfoque de reconfiguración centralizado o un enfoque descentralizado. Por un lado, en el enfoque centralizado [12][17] una entidad global describe todas las reglas de reconfiguración del sistema: tanto las reglas del propio sistema como las reglas de reconfiguración de los distintos subsistemas. Este enfoque tiene la ventaja de que todo el proceso de reconfiguración del sistema se encuentra centralizado, facilitando el mantenimiento de los distintos procesos de reconfiguración. Sin embargo, este enfoque no es apropiado, ya que presupone que la arquitectura de cada uno de los subsistemas es accesible para el configurador global (por lo que se viola el principio de encapsulación) y reconfigurable (no todos los subsistemas necesitarán ser reconfigurados). Por otro lado, siguiendo un enfoque descentralizado [25], tanto las reglas como los mecanismos de reconfiguración se describen dentro de cada subsistema. Esto tiene la ventaja de que cada subsistema determinará si es reconfigurable o no y cómo debe reconfigurarse, por lo que se respeta el principio de encapsulación y se facilita el mantenimiento de cada subsistema.

En este trabajo se presenta una propuesta para describir la reconfiguración programada de sistemas software de un modo descentralizado, con el objetivo de poder diseñar sistemas capaces de reconfigurarse, que a su vez puedan estar formados por otros subsistemas reconfigurables. La propuesta se basa en la identificación de la funcionalidad necesaria para proporcionar soporte a la reconfiguración dinámica, y su encapsulación en aspectos, con el objetivo de beneficiarse de las ventajas derivadas del enfoque AOSD, como la reutilización y la facilidad de mantenimiento.

Es por esta razón que la propuesta de reconfiguración dinámica se ha desarrollado sobre el modelo PRISMA [27], dado que permite la descripción de arquitecturas software orientadas a aspectos, y su compilación posterior a código C#, ejecutable en el middleware PRISMANET [26].

2.1. PRISMA

PRISMA [27] es una propuesta para el desarrollo de arquitecturas software orientadas a aspectos independientes de tecnología. Esta propuesta se caracteriza por definir un modelo de aspectos simétrico [19], en el que toda la funcionalidad de un elemento arquitectónico se encapsula mediante aspectos. Un aspecto representa un conjunto de funcionalidades comunes (*concern*) -seguridad, coordinación, reconfiguración, etc.-, que puede ser importado por aquellos elementos arquitectónicos que necesiten incorporar el comportamiento del "concern" que dicho aspecto define. Como resultado, un elemento arquitectónico es definido por un conjunto de aspectos que describen las distintas facetas del elemento arquitectónico.

Cada elemento arquitectónico PRISMA encapsula su funcionalidad como una caja negra y publica un conjunto de servicios que ofrecen a otros elementos arquitectónicos. Estos servicios, agrupados en interfaces, son publicados a través de los puertos, que son los puntos de interacción entre elementos arquitectónicos. Sin embargo, la

vista interna de dichos elementos arquitectónicos difiere entre simples (componentes y conectores) y complejos (sistemas).

Por un lado, la vista interna de un elemento arquitectónico simple es una composición invasiva de aspectos [1], que puede ser vista como un prisma (ver Figura 1, izquierda). Cada lado del prisma es un aspecto que el elemento arquitectónico importa. Los aspectos son sincronizados entre ellos a través de *weavings*, que indican cómo la ejecución del servicio de un aspecto activará la ejecución de servicios en otros aspectos.

Por otro lado, la vista interna de un elemento arquitectónico complejo incluye un conjunto de elementos arquitectónicos (componentes, conectores y otros sistemas) y las conexiones entre ellos (ver Figura 1, derecha). Estas conexiones pueden ser de dos tipos: *attachments*, que modelan las comunicaciones realizadas dentro del sistema (entre componentes y conectores); y *bindings*, que modelan las comunicaciones desde o hacia el exterior del sistema.

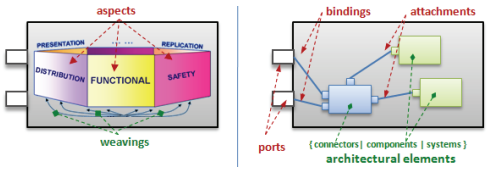

Figura 1. Vista interna y externa de un elemento arquitectónico PRISMA

2.2. Reconfiguración Dinámica en PRISMA: el Componente Evolver

En PRISMA, la capacidad de reconfiguración dinámica se ha modelado a través de un componente especial, denominado *Evolver*. Este componente proporciona los servicios y mecanismos necesarios de reconfiguración, y es importado por cada sistema que requiera ser reconfigurable en tiempo de ejecución. De esta forma, el compilador de modelos PRISMA puede determinar qué sistemas tienen que generarse con la infraestructura necesaria para poder ser reconfigurados dinámicamente y cuáles no.

El componente Evolver encapsula el *concern* de reconfiguración dinámica del sistema al que pertenece. El concern de reconfiguración dinámica es proporcionado por cuatro aspectos distintos y complementarios (ver Figura 2), los cuales se describen brevemente a continuación.

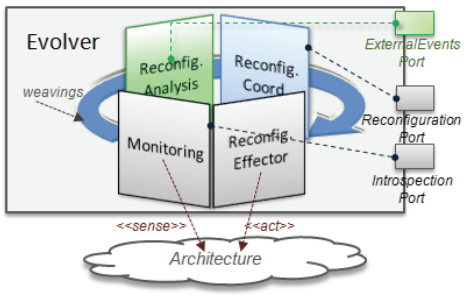

Figura 2. El componente Evolver y los aspectos de Reconfiguración

El aspecto *Reconfiguration Analysis* describe el conjunto de reconfiguraciones programadas a realizar en una arquitectura específica (aquella donde ha sido importado el Evolver) y el conjunto de eventos que activarán el proceso de reconfiguración.

El aspecto *Reconfiguration Coordination* proporciona los servicios de reconfiguración disponibles, verifica que se cumplan las restricciones estipuladas en el tipo del sistema (i.e. qué reconfiguraciones pueden realizarse y cuáles no), y coordina la ejecución transaccional de los servicios de reconfiguración para así garantizar que las reconfiguraciones se realizan de forma segura y en caso de que algo falle, devolver el sistema a su estado previo. Este aspecto garantiza que sólo los elementos arquitectónicos que van a ser reconfigurados sean detenidos de forma segura (i.e. alcanzando un estado quiescente), y que sólo cuando los elementos afectados por la reconfiguración estén preparados, se realice la reconfiguración.

El aspecto *Monitoring* proporciona servicios para monitorizar la arquitectura y recoger información relevante para activar el proceso de reconfiguración. Este aspecto proporciona tres tipos de información: (i) eventos ocurridos en la arquitectura (e.g. la invocación de servicios); (ii) información relativa a la configuración actual, lo que permite al sistema ser consciente de su estado: cuáles son sus instancias arquitectónicas actualmente en ejecución y cómo están conectadas; y (iii) información interna de los elementos en ejecución, como el estado de

ejecución: *idle* (parado, a la espera), *active* (está procesando servicios) o *blocked* (detenido temporalmente, listo para reconfigurar).

Por último, el aspecto *Reconfiguration Effector* es el que proporciona los servicios dependientes de tecnología para cambiar la arquitectura del sistema en tiempo de ejecución: (i) servicios para parar/iniciar la ejecución de los elementos arquitectónicos, (ii) servicios para crear/destruir instancias, (iii) servicios para crear/destruir conexiones y (iv) servicios para transferir el estado.

Los aspectos Monitoring, Reconfiguration Effector y Reconfiguration Coordination encapsulan los mecanismos genéricos de reconfiguración. Estos aspectos son predefinidos en el metamodelo de PRISMA y son importados por todos los sistemas reconfigurables. De este modo, el código de reconfiguración es centralizado en tres aspectos, facilitando su mantenimiento y reutilización. Por su parte, el aspecto Reconfiguration Analysis describe cómo debe reconfigurarse la arquitectura en la cual el Evolver ha sido importado, por lo que es distinto para cada sistema. Estos aspectos se sincronizan entre sí a través de weavings, los cuales se especifican fuera de los aspectos, en la definición del componente Evolver. Esto hace a los aspectos independientes entre sí, puesto que sus dependencias están aisladas en los weavings.

Por otro lado, el hecho de que el Evolver sea un elemento más de la arquitectura del sistema permite conectarlo con cualquier elemento arquitectónico. Por ejemplo, el puerto *externalEventsPort* (ver Figura 2) permite enviar eventos definidos por el usuario al aspecto Reconfiguration Analysis, y con ello activar el proceso de reconfiguración. Esto tiene sentido cuando se desea reconfigurar una arquitectura a consecuencia de eventos procedentes del exterior del sistema o a consecuencia de un evento específico generado por un componente interno al sistema. Detalles adicionales sobre el Evolver y los aspectos de reconfiguración pueden encontrarse en [10].

3. AgroBot: Uso de la Reconfiguración para la Recuperación Frente a Fallos

La agricultura es un sector especialmente relevante en la economía del levante español. La robótica de servicios [16], [21] puede jugar un papel fundamental para abordar las dificultades del sector: costes elevados de mano de obra, competencia de países en vías de desarrollo, falta de agua, preocupación medioambiental, control de plagas, etc. Existen granjas inteligentes en las que los robots realizan misiones de interés para la investigación. Esto constituye un banco de pruebas excelente, ya que es posible empezar por casos de estudio muy sencillos y cercanos a la robótica industrial (caracterizados por la realización de tareas repetitivas) e ir aumentando progresivamente la complejidad, siendo dotados de mayor autonomía.

AgroBot es un robot agrícola cuyo objetivo es la supervisión de un área determinada para controlar la presencia de plagas y/o la falta de agua. El sistema de visión artificial juega un papel muy importante para la realización autónoma de tareas, como es el movimiento por el terreno y la inspección de plagas. Este sistema es un elemento crítico en el que debe incorporarse tolerancia a fallos: si éste deja de funcionar, el robot deja de ser autónomo y por tanto deja de ser útil.

Este sistema incorpora una tarjeta capturadora de vídeo y un componente hardware para el procesamiento de las imágenes capturadas. Dicho componente implementa algoritmos específicos para el procesamiento de vídeo, optimizados para procesar la información que necesita el robot. Su implementación se ha realizado en hardware por motivos de rendimiento: los algoritmos implementados directamente en hardware se ejecutan más rápidamente que los implementados en software.

Una forma de proporcionar tolerancia a fallos en el sistema de visión sería a través de la redundancia de los componentes críticos: introducir varias réplicas de la tarjeta capturadora de vídeo, introducir elementos de votación o sumadores para unificar los resultados, etc. Sin embargo, tener más de un componente de procesamiento de vídeo ejecutándose en el sistema es muy costoso, dado que dichos algoritmos consumen muchos recursos.

Una alternativa es dotar al sistema de visión de una implementación *software* del componente de procesamiento de imágenes, que aunque con menor rendimiento, puede instanciarse en tiempo de ejecución a través de la reconfiguración dinámica, permitiendo al sistema recuperarse de

un eventual fallo del componente en su versión *hardware*. A continuación se describe cómo se ha especificado la estrategia de recuperación frente a fallos del sistema de visión del AgroBot, concretamente, para recuperarse de un fallo en el componente de procesamiento de imágenes.

3.1. Arquitectura software del sistema de visión del AgroBot

El sistema AgroBot ha sido especificado en el lenguaje de descripción de arquitecturas de PRISMA, con el objetivo de poder definirlo a un alto nivel de abstracción y posteriormente generar automáticamente el código de la plataforma destino. La arquitectura software del sistema de visión del AgroBot es la que se muestra en la Figura 3.

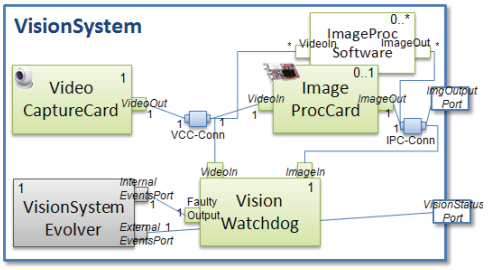

Figura 3. Arquitectura PRISMA del sistema de visión del AgroBot

El componente *VideoCaptureCard* encapsula a la tarjeta capturadora de imágenes. Dicho componente, a través del puerto, envía la secuencia de imágenes capturadas tras un intervalo de tiempo determinado. El componente *ImageProcCard* encapsula a la tarjeta procesadora de imágenes hardware, que procesa las imágenes recibidas de la tarjeta gráfica y extrae la información necesaria que será analizada por el resto de subsistemas del AgroBot. La información extraída de las imágenes es proporcionada al resto de subsistemas a través del puerto *ImgOutputPort*.

El componente *VisionWatchdog* monitoriza la ejecución de los componentes *VideoCaptureCard* e *ImageProcCard*, para comprobar que el funcionamiento es el esperado. Existen distintas estrategias de monitorización, de las cuales se ha elegido una variante del patrón *monitor-actuator* [14]: el monitor recibe parte de los datos de entrada del componente monitorizado y los procesa lo suficiente para compararlos con los datos de salida del componente monitorizado y detectar si éstos están dentro de unos márgenes de error. La ventaja de esta estrategia de monitorización es que es relativamente fácil de implementar y consume pocos recursos de computación, ya que el algoritmo para generar el resultado esperado del componente monitorizado es sencillo. Si el componente *VisionWatchdog* detecta un fallo (los componentes monitorizados no envían datos o los envían fuera de los márgenes de error), este componente genera un evento "FaultyOutput" indicando el tipo de componente que ha fallado.

El componente *ImageProcSoftware* proporciona una implementación alternativa (vía software) al componente *ImageProcCard*, y se encuentra inicialmente deshabilitado (en blanco).

Estos cuatro componentes se encuentran conectados a través de los conectores VCC-Conn y IPC-Conn, respectivamente (en azul en la figura). Dichos conectores coordinan el envío y recepción de imágenes entre los componentes.

Por último, el componente *VisionSystem Evolver* proporciona al sistema de visión del AgroBot el soporte a la reconfiguración dinámica. Es en este componente donde se especifica cómo debe reconfigurarse el sistema, y es por tanto donde se han descrito las estrategias de recuperación frente a fallos. A través del puerto *ExternalEventsPort* (ver Figura 3), este componente recibe del componente *VisionWatchdog* la notificación de cuándo ha habido un fallo, activándose de este modo el proceso de recuperación frente al fallo notificado.

3.2. Reconfiguración dinámica del sistema de visión del AgroBot

El componente VisionSystemEvolver importa los siguientes aspectos: (i) los aspectos predefinidos Reconfiguration Coordination, Monitoring y Reconfiguration Effector (ver sección 2.2), que son los que proporcionan los mecanismos de reconfiguración; y (ii) el aspecto VisionSystemEvolution, que es un aspecto del tipo Reconfiguration Analysis (ver sección 2.2) y describe cómo debe reconfigurarse el sistema de visión del AgroBot ante determinados eventos (como la presencia de fallos en el sistema).

Los aspectos de tipo Reconfiguration Analysis se dividen en dos secciones: un conjunto de transacciones de configuración y un conjunto de triggers (disparadores) de reconfiguración. Una transacción de configuración es una especificación que describe un conjunto de operaciones de (re)configuración que deben llevarse a cabo transaccionalmente (todo o nada) para alcanzar una nueva configuración válida. De este modo, las solicitudes de servicios de configuración se ejecutarán secuencialmente, y si algo falla, la configuración será deshecha. Un trigger de reconfiguración es una condición que, si se evalúa a cierta, activa una transacción de configuración. Una condición puede utilizar: (a) la información proporcionada por el aspecto Monitoring, con la que puede comprobarse si un servicio ha sido invocado en la arquitectura, si una conexión ha sido creada o destruida, etc.; y/o (b) eventos externos o definidos por el usuario, como es el caso de la detección de un fallo por parte del componente VisionWatchdog.

En la Figura 4 se muestra un fragmento del aspecto VisionSystemEvolution, en el cual se describe cómo reconfigurar la arquitectura del sistema de visión tras detectar el fallo del componente ImageProcCard. La transacción *repairImageProcessingUnit* (ver Figura 4, sección "transactions") define cómo debe reemplazarse el componente ImageProcCard por el componente ImageProcSoftware. El primer paso (ver proceso `PREPARE`) es obtener las referencias a las instancias que van a ser modificadas: el componente ImageProcCard que está fallando, y los conectores que lo enlazan al resto de elementos del sistema VCC-Conn y IPC-Conn. El siguiente paso es la reconfiguración (ver proceso `RECONF` en Figura 4): (1) se instancia el componente software para el procesamiento de imágenes; (2) se crea un attachment con el conector de la tarjeta capturadora de imágenes y (3) el conector que exporta los resultados; (4, 5) se destruyen las conexiones entre el componente con fallos y los respectivos conectores; y (6) se destruye la instancia del componente ImageProcCard, lo que implica desconectar físicamente el recurso hardware asociado (la tarjeta procesadora) para evitar que siga consumiendo energía del robot. Finalmente, el proceso END (7) indica el fin de la transacción de reconfiguración, para que se confirmen los cambios si todo ha ido correctamente.

La activación de este proceso es realizada cuando se recibe el evento "FaultyOutput", junto al tipo del componente que está fallando, *failingComponent*, por parte del componente VisionWatchdog. Este hecho se describe en la sección de *triggers* (ver Figura 4).

```
Reconfiguration Analysis aspect
     VisionSystemEvolution
...
Transactions
RepairImageProcessingUnit():
  PREPARE ::=
    oldImProcCardID=imageProcCard-list[0] -->
    VCCConnID=VCC-Conn-list[0] -->
    IPCConnID=IPC-Conn-list[0] --> RECONF;
  RECONF ::=
1   create-ImageProcSoftware!(cameraPos,
              output newImProcID) -->
2   attach-Att VCCConn IPCSW!(VCCConnID,
        newImProcID, output newAttID) -->
3   attach-Att_IPCSW_IPCConn!(newImProcID,
        IPCConnID, output newAttID) -->
4   detach-Att_VCCConn_IPC!(VCCConnID,
        oldImProcCardID) -->
5   detach-Att IPC IPCConn!(oldImProcCardID,
        IPCConnID) -->
6   destroy-ImageProcCard!(oldImProcCardID)-->
7   END;
...
Triggers
RepairImageProcessingUnit() when
  {failingComponent="ImageProcCard"}
     faultyOutput?(failingComponent);
...
End_Aspect VisionSystemEvolution;
```

Figura 4. Especificación del proceso de recuperación frente al fallo en el componente *ImageProcCard*

La ventaja de este enfoque es que, además de evitar el uso de la redundancia para la tolerancia a fallos, dota al sistema también de mecanismos para poder adaptarse a cualquier cambio en tiempo de ejecución: por ejemplo, reorganizar su arquitectura para que, en condiciones de baja energía, usar menos componentes o replicar aquellos que están siendo utilizados de una forma intensiva. Además, las reconfiguraciones programadas definidas en el aspecto Reconfiguration Analysis también pueden ser modificadas en tiempo de ejecución para permitir al sistema adaptarse a cambios no previstos inicialmente. Esto es realizado a través de la sustitución dinámica de aspectos [8], [9].

4. Trabajos relacionados

En los últimos años, numerosas propuestas han abordado el problema de la reconfiguración dinámica de sistemas software [3], [4], [11]. Sin embargo, pocos trabajos han tratado a la reconfiguración dinámica como un *crosscutting-concern*, con lo que con frecuencia el código de reconfiguración tiende a mezclarse con el código funcional. Rasche y Polze [30] utilizan aspectos para separar el código funcional del código de configuración del sistema, definiendo unos servicios de reconfiguración básicos. Otros trabajos, como Trap/J [31] o Cámara et al. [6] han utilizado el desarrollo orientado a aspectos para soportar la adaptación dinámica del software, a través de la agregación dinámica de aspectos. Sin embargo, estos trabajos están orientados hacia el uso de una tecnología concreta, mientras que nuestra propuesta se orienta a un nivel de abstracción mayor, las arquitecturas software.

En el área de las arquitecturas software, la reconfiguración dinámica puede proporcionarse de forma centralizada o de forma descentralizada. Por un lado, los enfoques centralizados tienen en común el uso de una entidad global que describe y gestiona cómo debe reconfigurarse todo el sistema. Los trabajos de Dashofy [12] y Garlan [17] proporcionan soporte a la reconfiguración dinámica programada, y la utilizan para describir la auto-reparación del sistema, basándose en modelos que describen la arquitectura válida del sistema. El inconveniente de estos trabajos es precisamente la no descentralización de la reconfiguración, bajo la suposición de que todos los subsistemas deben ser reconfigurables y accesibles. Por otro lado, la mayoría de enfoques descentralizados tienen en común el soporte de la reconfiguración dinámica a través de primitivas específicas del lenguaje, como LEDA [7], PiLaR [11], Plastik [2] o SOFA [5]. Estas primitivas pueden utilizarse en cualquier elemento arquitectónico para describir cuándo y cómo debe reconfigurarse la arquitectura del sistema. Sin embargo, esto provoca que el código correspondiente a la reconfiguración se entremezcle con el código funcional, dificultando así su mantenimiento. Morrison [25] describe una propuesta de evolución descentralizada muy cercana a nuestro trabajo, y aunque el código de evolución no se ha encapsulado en aspectos, sí ha sido separado convenientemente del código funcional del subsistema al que pertenece.

Existen algunos trabajos en los que se ha utilizado la reconfiguración dinámica para diseñar sistemas tolerantes a fallos, aunque tan sólo se ha realizado de forma parcial o incompleta. Yurcik y Doss [29] describen las ventajas de utilizar la reconfiguración para la tolerancia a fallos, y con ello minimizar el uso de la redundancia, pero no describen cómo soportar la reconfiguración dinámica ni cómo facilitar el mantenimiento de dichos sistemas. En las arquitecturas Simplex [18], la reconfiguración se usa tan sólo para cambiar conexiones: si un elemento falla, el sistema se reconfigura para pasarle el control a otra de las réplicas que estaban en ejecución. Sin embargo, la descripción de la reconfiguración es dependiente de la tecnología y no se desarrolla la capacidad de crear dinámicamente nuevas instancias (cuya creación puede abstraer el hecho de activar físicamente un componente hardware que estaba instalado pero no en ejecución). De Lemos [13] describe un enfoque para la tolerancia a fallos basado en la gestión de excepciones y en la reconfiguración dinámica. En esta propuesta los fallos generan excepciones, que son manejadas por los elementos arquitectónicos, indicando cómo deben ser reconfigurados. El código de la reconfiguración se encuentra definido dentro de cada elemento arquitectónico, pero separado del código funcional, facilitando su mantenimiento. Sin embargo, no se describe cómo el sistema puede obtener información acerca de su configuración ni los mecanismos necesarios para ello.

5. Conclusiones y trabajos futuros

En este trabajo se ha presentado una aplicación de la reconfiguración dinámica de arquitecturas software para abordar el diseño de sistemas tolerantes a fallos.

Una de las contribuciones de esta propuesta es la encapsulación de toda la funcionalidad relativa a la reconfiguración dinámica en distintos aspectos: (i) Monitoring, que supervisa la arquitectura y proporciona información sobre la configuración actual del sistema; (ii) Reconfiguration Analysis, que define cuándo debe activarse el proceso de reconfiguración y cómo debe realizarse; (iii) Reconfiguration

Coordination, que coordina el proceso de reconfiguración para que se realice de una forma segura, y (iv) Reconfiguration Effector, que proporciona los servicios que modifican la arquitectura. De esta forma: (i) el código de reconfiguración se mantiene aislado del código funcional, (ii) las políticas de reconfiguración se mantienen separadas de los servicios que las llevan a cabo, y (iii) la reconfiguración se describe a un alto nivel de abstracción, en términos de la arquitectura que debe configurarse.

Esta propuesta se ha aplicado a un robot agrícola para describir, de modo independiente de tecnología, cómo el subsistema de visión debe recuperarse ante el fallo de uno de sus componentes y reemplazarlo en tiempo de ejecución por otro componente alternativo. El hecho de usar la reconfiguración dinámica en lugar de técnicas de redundancia permite no sólo reducir la complejidad de la arquitectura del robot, sino que además le dota de mecanismos para poder adaptarse a cualquier cambio en tiempo de ejecución: por ejemplo, reorganizar su arquitectura para que, en condiciones de baja energía, utilizar menos componentes o replicar aquellos que están siendo utilizados intensivamente.

Sin embargo, en un sistema robótico puede haber sistemas críticos cuya tolerancia fallos deba ser necesariamente implementada a través de la redundancia. Uno de ellos es el propio sistema de reconfiguración: como puede fallar, también necesita ser replicado. El otro caso son aquellos subsistemas en los que el tiempo de respuesta es crítico: la reconfiguración necesita un tiempo mínimo para aplicar las políticas de recuperación, mientras que con la replicación la respuesta es inmediata, pues existe un conjunto de réplicas que ya estaban proporcionando la misma funcionalidad del subsistema que ha fallado. El análisis de esta problemática se deja para trabajos posteriores.

Agradecimientos

Este trabajo ha sido financiado por el CICYT (Comisión Interministerial de Ciencia y Tecnología) proyecto MULTIPLE (TIN2009-13838).

Referencias

[1] Aßmann, U. Invasive Software Composition. Springer, 2003.

[2] Batista, T., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. 2nd European Workshop on Software Architectures (EWSA'05). LNCS, vol. 3527. Springer, 2005.

[3] Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. First Workshop on Self-Managed Systems (WOSS'04).

[4] Buckley, J., Mens, T., Zenger, M., et al.: Towards a taxonomy of software change. Journal on Software Maintenance and Evolution 17(5). Wiley, 2005.

[5] Bures, T., Hnetynka P., Plasil F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. 4th Int. Conf. on Software Eng. Research, Management and Applications (SERA'06). Seattle, WA, 2006.

[6] Cámara, J., Canal, C., Cubo, J., Murillo, J.M.: Enabling Adaptivity in User Interfaces. 1st European Conf. on Software Architecture (ECSA'07). LNCS,vol.4758. Springer, 2007.

[7] Canal, C., Pimentel, E., Troya, J.M.: Specification and Refinement of Dynamic Software Architectures. Working IFIP Conf. on Software Architecture (WICSA'99). San Antonio, Texas, USA, 1999.

[8] Costa, C., Pérez, J., Carsí, J.A: Dynamic Adaptation of Aspect-Oriented Components. 10th Int. Symp. on Component-Based Software Engineering (CBSE'07). LNCS, vol. 4608. Springer, 2007.

[9] Costa-Soria C., Hervás-Muñoz D., Pérez J., Carsí J.A. A Reflective Approach for Supporting the Dynamic Evolution of Component Types. 14th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'09). Potsdam, Germany, 2009.

[10] Costa-Soria C., Pérez J., Carsí J.A. An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures. 2nd Workshop on Autonomic and SELF-adaptive Systems (WASELF'09). San Sebastián, Spain, 2009.

[11] Cuesta, C.E., Fuente, P.d.l., Barrio-Solárzano, M.: Dynamic Coordination

Architecture through the use of Reflection. ACM Symp. on Applied Computing (SAC'01). USA, 2001.

[12] Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards Architecture-Based Self-Healing Systems. First Workshop on Self-Healing Systems (WOSS'02), 2002.

[13] de Lemos, R.: Architectural reconfiguration using coordinated atomic actions. Intern. Workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06). ACM, 2006

[14] Douglass, B.P.: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley Object Technology, 1999.

[15] Endler, M., & Wei, J.: Programming Generic Dynamic Reconfigurations for Distributed Applications. 1st International Workshop on Configurable Distributed Systems, 1992.

[16] Fernández, C., Iborra, A., Álvarez, B., et al.: Ship Shape in Europe: Co-operative Robots in the Ship Repair Industry. Robotics and Automation Magazine, 12(3). IEEE, 2005.

[17] Garlan, D., Cheng, S., Huang, S., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer, 37. IEEE 2004.

[18] German-Rivera, J., et al.: An Architectural Description of the Simplex Architecture, Tech. Report CMU/SEI-96-TR-006, 1996.

[19] Harrison, W. H., Ossher, H. L., Tarr, P. L.: Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Tech. Rep. RC22685 (W0212-147), T. J. Watson Research Centre, IBM, 2002.

[20] Heimerdinger, W.L., Weinstock, C.B.: A Conceptual framework for System Fault Tolerance. Tech. Report CMU/SEI-92-TR-033, 1992.

[21] Iborra, A., Pastor, J.A., Álvarez, B., Fernández, C., Fernández, J.M.: Robots in Radioactive Environments. Robotics and Automation Magazine, 10(4), 2003.

[22] Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-Oriented Programming. In: ECOOP'97. LNCS, vol. 1241. Springer, 1997.

[23] Kramer, J., Magee, J.: Dynamic configuration for distributed systems. IEEE Trans. Software Engineering, 11(4), 1985.

[24] McKinley, P.K, Sadjadi, S.M., Kasten, E.P, and Cheng, B.H.C.: Composing Adaptive Software. Computer, 37(7):56-64. IEEE,2004.

[25] Morrison, R., Balasubramaniam, D., Kirby,G., et al: A Framework for Supporting Dynamic Systems Co-Evolution. Autom. Software Eng. 14(3). Springer, 2007.

[26] Pérez, J, Ali, N, Costa, C, Carsí, J.A., Ramos, I.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. 3rd Int. Conference on .NET Technologies. Pilsen, Czech Republic, 2005.

[27] Pérez, J. PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Universidad Politécnica de Valencia, 2006.

[28] Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. 9th Symp. on Component-Based Software Engineering (CBSE06). LNCS, vol. 4063. Springer, 2006.

[29] Perry, D. E., & Wolf, A. L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Eng. Notes, 17(4), 1992

[30] Rasche, A., Polze, A.: Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET. 6th Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'03), 2003.

[31] Sadjadi, SM, McKinley, PK, Cheng, BHC, Kurt, RE: TRAP/J: Transparent generation of adaptable java programs. Int. Symp. on Distributed Objects and Applications (DOA'04). Agia, Cyprus, 2004.

[32] Yurcik, W., Doss, D.: Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration. IEEE Software, 18(4), 2001.